Launch Time Performance Guidelines

(Legacy)

Contents

2

Figures, Tables, and Listings

Introduction to Launch Time Performance Guidelines

> **Important:** This document may not represent best practices for current development. Links to downloads and other resources may no longer be valid.

This document provides specific tips on how to identify and improve your application's performance at launch time. The launch of an application provides users with the first impression of your application. If your application launches quickly, it leaves users with a good impression.

Organization of This Document

This document contains the following articles:

- Launch Time Tips (page 5) describes some standard techniques for improving the performance of your application at launch time.

- Gathering Launch Time Metrics (page 9) describes ways to measure the speed of your application launch.

- Minimizing File Access At Launch Time (page 14) shows you how to measure and reduce your file usage at launch time.

- Prebinding Your Application (page 17) explains the prebinding process and shows you how to determine if your application is prebound.

Launch Time Tips

Launch time is an important metric to measure for any application. It is the first experience a user has with your application, and it is something the user sees on a regular basis. The less time it takes your application to launch, the faster your application will seem to the user.

Your overriding goal during launch should be to display the application's menu bar and main window and then start responding to user commands as quickly as possible. Making your application responsive to commands quickly provides a better experience for the user. The following sections provide some general tips on how to make your application launch faster.

Regardless of what techniques you choose to improve your launch times, the only way to know you've improved performance is through measurement. Gather and record launch time metrics early in your development process and monitor them especially when implementing new optimizations. For information on how to measure launch-time performance, see Gathering Launch Time Metrics (page 9).

Delay Initialization Code

Many applications spend a lot of time initializing code that isn't used until much later. Delaying the initialization of subsystems that are not immediately needed can speed up your launch time considerably. Remember that the goal is to display your application interface quickly, so try to initialize only the subsystems related to that goal initially.

Once you have posted your interface, your application can continue to initialize additional subsystems as needed. However, remember that just because your application is able to process commands does not mean you need all of that code right away. The preferred way of initializing subsystems is on an as-needed basis. Wait until the user executes a command that requires a particular subsystem and then initialize it. That way, if the user never executes the command, you will not have wasted any time running the code to prepare for it.

For a Carbon application, you should perform your basic initialization before beginning your application's main event loop. Once that loop is running, you can set up a one-shot timer to execute any additional code that your application absolutely requires for basic operation. Do not load code for specific features until the user chooses a command that uses that feature.

For a Cocoa application, avoid putting a lot of extraneous initialization code in your `awakeFromNib` methods. The system calls the `awakeFromNib` method of your main nib file before your application enters its main event loop. Use that method to initialize the objects in that nib and to prepare your application interface. For all other initialization, use the `applicationDidFinishLaunching` method of your NSApplication object instead. For more information on nib files and how they are loaded, see *Loading Resources*.

Simplify Your Main Nib File

Loading a nib file is an expensive process that can slow down your application launch time if you are not careful. When a nib file is loaded, all of the objects in that file are instantiated and made ready for use. The more objects you include in your application's main nib, the more time it takes to load that file and launch your application.

The instantiation process for objects in a nib file requires that any frameworks used by those objects must themselves be resident in memory. Thus loading a nib for a Cocoa application would likely require the loading of both the AppKit and Foundation frameworks, if they were not already resident in memory. Similarly, if you declare a custom class in your main nib file and that class relies on other frameworks, the system must load those frameworks as well.

When designing your application's main nib file, you should include only those objects needed to display your application's initial user interface. Usually, this would involve just your application's menu bar and initial window, if any. For any custom classes you include in the nib, make sure their initialization code is as minimal as possible. Defer any time-consuming operations or memory allocations until after the class is instantiated.

Minimize Global Variables

For both applications and frameworks, be careful not to declare global variables that require significant amounts of initialization. The system initializes global variables before calling your application's `main` routine. If you use a global variable to declare an object, the system must call the constructor or initialization method for that object during launch time. In general, it's best to avoid declaring objects as global variables altogether when you can use a pointer instead.

If you are implementing a framework, or any type of reusable code module for that matter, you should also minimize the number of global variables you declare. Each application that links to a framework acquires a copy of that framework's global variables. These variables might require several pages of virtual memory, which then increases the memory footprint of the target application. An increased memory footprint can lead to paging in the application, which has a tremendous impact on performance.

One way to minimize the global variables in a framework is to store the variables in a malloc-allocated block of memory instead. In this technique, you access the variables through a pointer to the memory, which you store as a global variable. Another advantage of this technique is that it allows you to defer the creation of any global variables until the first time they are actually used. See Optimizing Your Memory Allocations in *Memory Usage Performance Guidelines* for more information.

Minimize Strings File Sizes

Loading large numbers of unused strings at launch time adds an unnecessary burden to your application's memory footprint. In low-memory situations, a larger footprint could trigger paging and impact the launch time of your application. If you find yourself loading hundreds of unused strings at launch time, you might consider using separate resource files to store those strings not needed to launch the application.

In general, minimizing the number of strings in your `Localizable.strings` file is a good idea and can improve your application launch time. You must be careful, though, not to break up your remaining strings files too much in an effort to minimize the number of strings loaded by each successive operation. Separating your strings into many small files increases the overall amount of time spent doing file I/O, which can hinder performance rather than improve it. Before breaking your strings files into more than a few files, you should gather metrics to make sure it is warranted.

Reduce the Impact of Expensive Operations

Application launch time is not the time to perform any operation involving a potentially large data set. If your application handles a scalable data set, make sure to gather performance metrics with a large data set. Even with efficient algorithms, processing large amounts of data takes time and should be deferred until after your application finishes launching.

If you must load data early, try to do so in a way that reduces the impact on your application's launch time. One way is to design your program in a way that lets you load only a portion of the data set. For a really large data set, the user will be unable to see it all on the screen at one time anyway. Loading and displaying data incrementally improves launch time as well as the general performance of your application. Another way is to use a background thread to load the data shortly after the launch cycle completes.

For more information about improving the speed of your operations, see *Code Speed Performance Guidelines*.

Avoid Memory Turnover

Allocating and deleting memory takes time. If you find your algorithms allocating and deleting temporary memory in a tight loop, you might think of a way to remove those allocation routines from the loop. Rather than create a new string object each time, you could create one string object and re-initialize its contents with each pass of the loop.

One way to determine if you're allocating too much memory at launch time is to launch your application running under MallocDebug or ObjectAlloc. These tools show you the allocation patterns of your application. If you find your application allocating large amounts of temporary memory, you might go back through your algorithms to see if there are any blocks you can reuse or recycle.

For information about the OS X virtual memory system and how to allocate memory efficiently, see *Memory Usage Performance Guidelines* .

Use Local Resources

At launch time, it's important to know where your resources are located. All of your application's critical resources should be located inside the application bundle itself. Searching for resources outside of the bundle has potentially serious costs, as you may not know whether the resource is local or on the network. In particular, you should keep in mind that items such as plug-ins, loadable bundles, and user preferences may reside somewhere out on the network. Attempting to load these resources at launch time can delay the availability of your application.

It's best to avoid accessing external resources at launch time. If a resource is on a network server and the network is not available, your application could hang while it waits for the needed resource to become available. This is not a desirable situation and should be avoided by eliminating startup dependencies on these types of resources. If you do need to load these resources early, try loading them after your application has finished launching or from a background thread.

Gathering Launch Time Metrics

The following sections describe some techniques for gathering launch-time performance metrics.

Measuring Launch Speed

One of the more important measurements you can make during your application launch cycle, is how long it takes before your application is ready to process user commands. The following sections describe several techniques for measuring the launch speed of your application.

Gathering Data Using Checkpoints

One way to gather information about your application's launch performance is to use checkpoints. Checkpoints let you bracket any block of code you want to monitor with identifying information. In the simplest case, you can insert a checkpoint at the beginning of your `main` function and again right after your initialization code finishes and record the time at which those checkpoints were encountered.

If you need a fine grain view of your application launch, you can write checkpoint code that retrieves the current time from the system and write it to a log file. If you do not need quite so much detail, you can implement a simpler form of checkpoint using file-system calls. In this technique, you call a function that touches the file system and then use `fs_usage` to record the time that call was made.

The following code snippet uses the `stat` system function as a checkpoint to mark the beginning of the launch cycle for the TextEdit application. The function attempts to touch a non-existent file, which in this case is just a string with the name of the checkpoint. The call itself fails but registers as an attempt to access a file and therefore shows up in the output from `fs_usage`.

```
struct stat statbuf;
stat("START:launch TextEdit", &statbuf);
```

With this code inserted into your application, you can then open a Terminal window and launch `fs_usage` with the `-w` option. You might want to redirect the output from `fs_usage` through the `grep` tool to report entries only from your application. For example, to report entries from the TextEdit application, you would use the following command:

```
% sudo fs_usage -w | grep TextEdit
```

With `fs_usage` running, launch your application. In the output from `fs_usage`, look for a `stat` call with the name of your checkpoint. For example, inserting the checkpoint "`START: launch TextEdit`" at the beginning of the TextEdit application yields output similar to the following:

```
14:13:59.689 stat    [ 2]    START:launch TextEdit   0.000081    TextEdit
```

You can then use the timestamp on the left to determine the amount of time elapsed between the two checkpoints. This information tells you the elapsed time taken to execute the code between those two checkpoints.

> **Note:** Be aware that the elapsed time between two checkpoints does not necessarily reflect the time spent executing your application code. It is simply an indication of how long it took to complete the task given the current activity level of the system.

For more information on using `fs_usage`, see Using fs_usage to Review File I/O (page 14).

Using Explicit Timestamps

One way to measure the speed of any operation, including launch times, is to use system routines to get the current time at the beginning and end of the operation. Once you have the two time values, you can take the difference and log the results.

The advantage of this technique is that it lets you measure the duration of specific blocks of code. OS X includes several different ways to get the current time:

- `mach_absolute_time` reads the CPU time base register and is the basis for other time measurement functions.

- The Core Services `UpTime` function provides nanosecond resolution for time measurements.

- The BSD `gettimeofday`function (declared in `<sys/time.h>`)provides microsecond resolution. (Note, this function incurs some overhead but is still accurate for most uses.)

- In Cocoa, you can create an NSDate object with the current time at the beginning of the operation and then use the `timeIntervalSinceDate:` method to get the time difference.

Measuring Cocoa Application Launch Times

If you are writing a Cocoa application, you can use hooks in the AppKit framework to shutdown your application immediately after it finishes launching. Setting the `NSQuitAfterLaunch` environment variable to any value causes a Cocoa-based application to exit immediately after completing its launch cycle. You can use this variable in conjunction with `fs_usage` to record the initial and final activity times of the application.

> **Important:** Setting the value of the `NSQuitAfterLaunch` environment variable to zero does not disable its effect. Instead, you must use the `unsetenv` command to remove the definition of this variable entirely.

Sampling the Application Launch

Sampling your application launch can identify where your application is spending its time. Sampling records which functions were called at regular intervals during your application's runtime. Using this data, you can identify operations that might be taking too much time and target them for optimization.
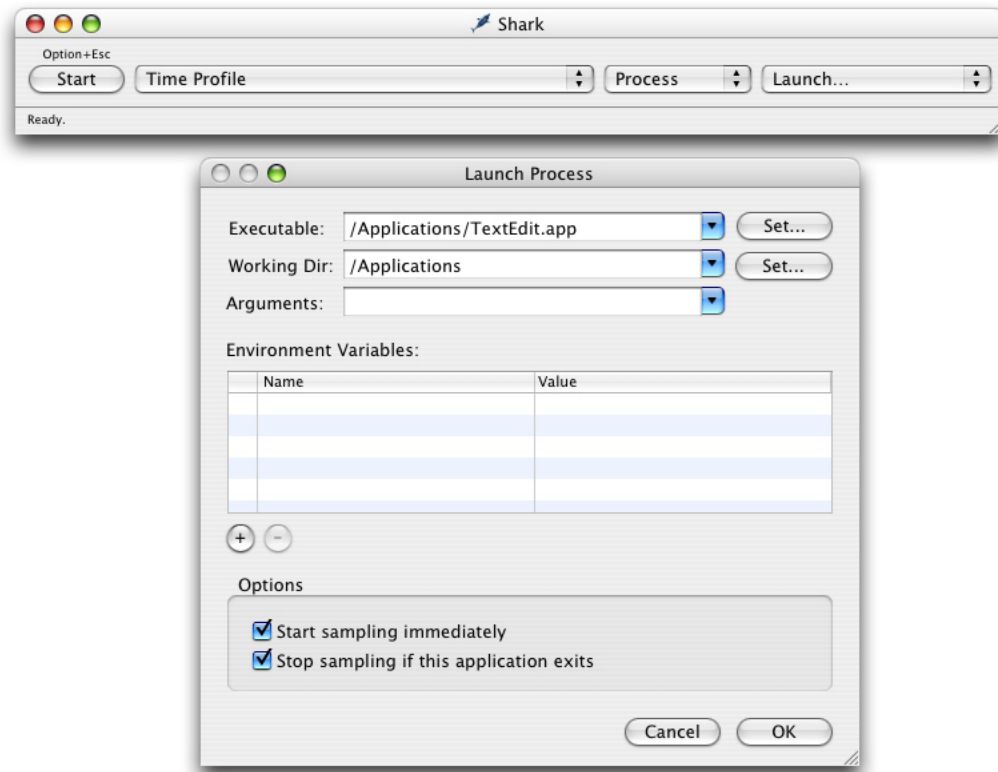
Using Shark

The Shark application provides a graphical interface for gathering call stack data at program launch time. To gather this data, do the following:

1. Launch Shark.

2. Set the sampling configuration to Time Profile.

3. Select Process from the target popup menu. Another popup menu appears with a list of running processes and a "Launch" option. (See Figure 1.)

4. Select the Launch option to display the Launch Process window.

5. From the Launch Process window, select the process you want to launch along with any arguments or environment variables you need to launch the program.

6. Make sure the "Start sampling immediately" check box is enabled.

7. Click OK to dismiss the Launch Process window. Shark immediately launches the selected process and begins sampling.

| 2014-03-10 | Copyright © 2003, 2014 Apple Inc. All Rights Reserved.

11

8.   When your application is done launching, click the Stop button (or use the Option+Esc hotkey) to stop sampling and view the results.

Figure 1   Sampling the launch of an application with Shark

In addition to gathering sample data, you can also use shark to trace specific function calls, including malloc calls, at launch time. For more information about Shark, see the Shark User Manual.

Using the sample Command-Line Tool

Another way to gather launch-time performance metrics is to use the `sample` command-line tool. Like Sampler, the `sample` tool periodically samples an application and creates a runtime graph of the functions that were called. You can use the sampled data to see get a more detailed view of what your application was doing during launch.

You must run `sample` with the `-wait` option to generate information for a launching application. The `-wait` option tells `sample` to wait for the existence of the process and to begin sampling it with the specified interval and duration when it appears. For example, you could use the following command to sample the launch of the TextEdit application for 5 seconds at 10 millisecond intervals.

```
sample TextEdit 5 10 —wait
```

When calling the `sample` tool, let the sampled application continue running until the sampling period is over. When the `sample` tool writes out its report, it uses the application's symbol table to identify the routines that were called. If you quit the application before the sampling period is over, the symbol information may become unavailable. You can also use the `—mayDie` option to try to locate the symbol information explicitly.

Minimizing File Access At Launch Time

Accessing a file is one of the slowest operations performed on a computer, so it is important that you do it as little as possible, especially at launch time. There is always some file access that must occur at launch time, such as loading your executable code and reading in your main nib file, but reducing your initial dependence on startup files can provide significant speed improvements.

Delay Any Unnecessary File I/O

If you can delay the reading of a file until after launch time, do so. The following list includes some files whose contents you may not need until after launch:

- Frameworks not used directly by your application—avoid calling code that uses non-essential frameworks until after launch.

- Nib files whose contents are not displayed immediately—make sure your nib files and `awakeFromNib:` code are not doing too much at launch time. See Simplify Your Main Nib File (page 6) for more information.

- User preference files—user preferences may not be local so read them later if you can.

- Font files—consider delaying font initialization until after the application has launched.

- Network files—avoid reading files located on the network if at all possible.

If you must read a file at launch time, do so only once. If you need multiple pieces of data from the same file, such as from a preferences file, consider reading all of the data once rather than accessing the file multiple times.

Using fs_usage to Review File I/O

One way to identify the files used by your application at launch time is with the `fs_usage` tool. To monitor launch-time activity from your application, start running `fs_usage` in a Terminal window before you launch your application. The tool generates a continuous stream of data regarding all file system accesses.

> **Important:** You must have root access to run `fs_usage`. You can use the `su` or `sudo` commands to run the tool.

To view file activity for all processes with the `fs_usage` tool, you would enter the following at the Terminal prompt.

```
% sudo fs_usage
```

If you wanted to limit the display to files accessed by a particular process, you could redirect the output through the `grep` tool. For example, to display file behavior for the TextEdit application, you would enter the following at the Terminal prompt:

```
% sudo fs_usage | grep TextEdit
```

After entering your root password, `fs_usage` begins running. There should be a flurry of activity generated by `fs_usage` when you launch your application. Once your application finishes launching, stop `fs_usage` by typing Control-C in your Terminal window.

Listing 1 shows a small portion of the output obtained during the launch of the TextEdit application. Pay attention to the second and fourth columns, which identify the operation and the elapsed time (in seconds) spent in that operation. You can generate additional info by passing the `-w` option to `fs_usage` or by maximizing the width of your Terminal window.

Listing 1   Sample output from fs_usage

```
10:56:13   CACHE_HIT                                               0.000041
   TextEdit
11:00:04   CACHE_HIT                                   0.000024   TextEdit
11:00:04   CACHE_HIT                                   0.000032   TextEdit
11:00:04   CACHE_HIT                                   0.000026   TextEdit
11:00:04   lstat          tions/TextEdit.app/Contents/MacOS  0.000052   TextEdit
11:00:04   lstat          tEdit.app/Contents/MacOS/TextEdit  0.000020   TextEdit
11:00:04   stat           /Applications/TextEdit.app         0.000012   TextEdit
11:00:04   access         /Applications/TextEdit.app         0.000008   TextEdit
11:00:04   lstat          ents/Resources/DocumentWindow.nib  0.000030   TextEdit
11:00:04   statfs         ents/Resources/DocumentWindow.nib  0.000019   TextEdit
11:00:04   open           ents/Resources/DocumentWindow.nib  0.000022   TextEdit
11:00:04   getdirentries                                      0.000067   TextEdit
```

```
11:00:04  getdirentries                                0.000005    TextEdit
11:00:04  close                                        0.000007    TextEdit
```

The preceding sample data shows how much time was spent getting information about the TextEdit binary file and its document window nib file. In this example, most of the operations took only microseconds to perform. You should search your own output to see if there are any files being accessed that aren't really needed, or whose access takes a significant amount of time.

Prebinding Your Application

Prebinding is the process of computing the addresses for symbols imported by a shared library or application prior to their use. Resolving these addresses before their use reduces the amount of work performed by the dynamic loader (`dyld`) at runtime and results in faster launch times for applications.

In OS X v10.4, `dyld` was improved in a way that eliminated the need for prebinding information in most situations. The system libraries are now the only executables that are still prebound, and they are prebound in a way that optimizes performance on the target system. Because of the improved prebinding of the system libraries, applications and third-party libraries no longer need to be prebound. A side benefit to this new behavior is that applications now have more usable address space than in previous versions of the operating system.

If you are developing applications for versions of OS X prior to v10.4, prebinding is considered optional. Changes in v10.3.4 made application prebinding unnecessary but applications running on earlier versions of the operating system still received some benefits from prebinding. If you feel your application launches slowly on pre-10.3.4 systems, build your application prebound and see if launch time improves.

If you are developing frameworks or other dynamic shared libraries for versions of OS X prior to 10.4, it is still recommended that you specify a base address for your library. Specifying this address allows prebinding to occur for applications that use your library. If your library is running in OS X v10.4 and later, specifying a base address is optional but can be useful for debugging shared libraries. The `atos` command-line tool lets you identify symbols located in memory. It is easier to identify specific symbols if your library has a known base address.

The following sections tell you how to prebind your application and framework projects and how to update that prebinding information if it becomes invalid.

Prebinding Your Code

Prior to OS X v10.4, prebinding was enabled for all new projects built using Xcode. In OS X v10.4 and later, this setting is no longer enabled due to changes that make prebinding unnecessary. If you aren't sure if your project is being built with prebinding enabled, you can check the build settings for your project. Prebinding settings are set on a per-target basis in the Build options view of the Xcode inspector window. If the "Prebinding" option is enabled for your target, it is being built with prebinding information.

If you are not using Xcode, there are several other ways to enable prebinding of your application or framework. During the link phase, the `ld` tool looks to see if the `LD_PREBIND` environment variable is set. If it is, the tool enables prebinding unless a command-line option specifically disables it. If you are calling `ld` from the command-line, you can pass it the `–prebind` option to enable prebinding explicitly.

If you are developing a framework for versions of OS X prior to 10.4, you should always build and ship it with prebinding enabled. If your framework is not prebound, applications that reference your framework cannot be prebound either, which can impact their launch time. In addition to enabling prebinding, you need to specify a preferred memory address for your framework. You do this by passing the `–seg1addr` option to `ld`. In Xcode, you add this option to the "Other Linker Flags" build setting. From the command line, simply include this option along with the other linker options in your makefile. For more information about using this option, see the `ld` man page.

Caveats for Prebinding

If you build with prebinding enabled, there are still times when `ld` may be unable to prebind your application. Prebinding fails if there are any symbol-name conflicts in the linked libraries or if the preferred address spaces for any libraries overlap. Prebinding also fails if any linked frameworks are not themselves prebound. When prebinding fails, the dynamic linker has to readjust the addresses of symbols in the affected libraries, which can slow down launch time.

In order to minimize symbol-name conflicts, Apple introduced a two-level namespace mechanism in OS X v10.0.4. Two-level namespaces use both the library name and the symbol name to identify each symbol, which reduces the chances of a collision. An executable built with the two-level namespace format is still compatible with versions of OS X prior to version 10.0.4. By default, Xcode builds all new projects using two-level namespaces.

Apple builds and ships its libraries with prebinding enabled and in a way that makes sure there are no address-space overlaps. However, other applications may install libraries and frameworks whose prebound addresses do overlap. Table 1 lists the virtual memory address ranges available to your code on versions of OS X prior to version 10.2. This table also lists the address ranges reserved for Apple-supplied frameworks and services. You can use this information to choose an appropriate location for your framework and library code.

Table 1   Prebinding address ranges prior to OS X v10.2)

Address range	Usage
`0x00000000` to `0x410FFFFF`	Application and user framework address range.
`0x41100000` to `0x412FFFFF`	Address range reserved for use by Apple frameworks. Do not use this address range for your libraries.

Address range	Usage
`0x41300000` to `0x606DFFFF`	Address range is preferred for use by Apple frameworks
`0x606E0000` to `0x6FFFFFFF`	Additional space available for third-party application and framework code.
`0x70000000` to `0x8FFFFFFF`	Address range reserved for use by Apple frameworks. Do not use this address range.
`0x90000000` to `0x9FFFFFFF`	Additional space available for third-party application and framework code.
`0xA0000000` to `0xAFFFFFFF`	Preferred for use by the Window Manager.
`0xB0000000` to `0xBFFFFFFF`	Preferred for use by thread stacks.
`0xC0000000` to `0xFEFFFFFF`	Additional space available for third-party application and framework code.
`0xFF000000` to `0xFFBFFFFF`	Preferred for use by the pasteboard and other system services.
`0xFFC00000` to `0xFFFFFFFF`	Additional space available for third-party application and framework code.

Table 2 lists the virtual memory address ranges available with OS X version 10.2 through version 10.3.x. In cases where an address range is preferred by Apple frameworks or services, you may still be able to use portions of that range. The availability of a given range depends on which frameworks or services your application uses.

Table 2   Prebinding address ranges for OS X v10.2 to v10.3.x

Address range	Usage
`0x00000000` to `0x4FFFFFFF`	Application address range.
`0x50000000` to `0x8FDFFFFF`	Preferred address range for Apple frameworks. Applications may use this range as necessary.
`0x8FE00000` to `0xAFFFFFFF`	Reserved for use by Apple frameworks. Do not use this address range.
`0xB0000000` to `0xBFFFFFFF`	Preferred address range for the application's main thread. The Window Manager may also use portions of this range. Applications may use this range as necessary.
`0xC0000000` to `0xEBFFFFFF`	Additional space available for third-party application and framework code.

Address range	Usage
0xEC000000 to 0xEFFFFFFF	Preferred for use by the Apple prebinding tools. Applications may use this range as necessary.
0xF0000000 to 0xFDFFFFFF	Preferred for use by additional thread stacks. Applications may use this range as necessary.
0xFE000000 to 0xFFBFFFFF	Reserved for use by the pasteboard and other system services. Do not use this address range.
0xFFC00000 to 0xFFFDFFFF	Preferred for use by other system services. Applications may use this range as necessary.
0xFFFE0000 to 0xFFFFFFFF	Reserved for use by system services. Do not use this address range.

Table 3 lists the virtual memory address ranges available with OS X version 10.4 and later. This version consolidates the system libraries into a single address range and frees up more contiguous space for your application to use.

Table 3   Prebinding address ranges for OS X v10.4 and later

Address range	Usage
0x00000000 to 0x8FDFFFFF	Application address range.
0x8FE00000 to 0xAFFFFFFF	Reserved exclusively for Apple system libraries. Do not use this address range.
0xB0000000 to 0xBFFFFFFF	Preferred address range for the application's main thread.
0xC0000000 to 0xEBFFFFFF	Additional space available for third-party applications and framework code.
0xEC000000 to 0xEFFFFFFF	Preferred for use by the Apple prebinding tools. Applications may use this range as necessary.
0xF0000000 to 0xFDFFFFFF	Preferred for use by additional thread stacks. Applications may use this range as necessary.
0xFE000000 to 0xFFBFFFFF	Reserved for use by the pasteboard and other system services. Do not use this address range.
0xFFC00000 to 0xFFFDFFFF	Preferred for use by other system services. Applications may use this range as necessary.

Address range	Usage
`0xFFFE0000` to `0xFFFFFFFF`	Reserved for use by system services. Do not use this address range.

An application's binary code is loaded beginning at address `0x00000000`. You should never define a framework with a low address range as it will very likely collide with the address range of any applications that use it. Instead, use an address range that is higher in the available address space.

Determining if Your Executable Is Prebound

The simplest way to determine if your Mach-O executable is prebound is to use the `otool` command-line tool to examine the object file. Running this tool with the `-h` and `-v` options displays the Mach header information for the executable. If your executable is prebound, you should see the word `PREBOUND` in the flags section of the header. The following code listing shows the output for the TextEdit application.

```
Mach header
      magic cputype cpusubtype    filetype ncmds sizeofcmds      flags
   MH_MAGIC      PPC         ALL     EXECUTE    54       8108    NOUNDEFS DYLDLINK
PREBOUND TWOLEVEL
```

Prior to OS X v10.4, another way to determine if a Mach-O executable is prebound is to enable the prebinding debugging option and launch your executable. (This does not work for applications in OS X v10.4 and later because prebinding for main executables is ignored.) From the `csh` shell, you can do this using the following steps:

1. Launch Terminal.

2. At the Terminal prompt, type the following:

   ```
   setenv DYLD_PREBIND_DEBUG
   ```

3. At the Terminal prompt, enter the path to your application's executable file. For TextEdit, you would enter something like the following:

   ```
   /Applications/TextEdit.app/Contents/MacOS/TextEdit
   ```

If your application is prebound, the `dyld` tool outputs the message `prebinding enabled` to the command line. If you see messages about some number of two-level prebound libraries being used, then your application is only partially prebound.

Fixing Prebinding Information

In all versions of OS X, you should not need to do anything to keep your prebinding information up-to-date on the user's system. On versions of OS X that require it, the system automatically fixes prebinding information as needed. In addition, the Installer program runs the `update_prebinding` tool at the end of the install cycle to update prebinding information. You should never need to call this tool directly.

Document Revision History

This table describes the changes to *Launch Time Performance Guidelines*.

Date	Notes
2014-03-10	Moved to Retired Documents Library.
2006-04-04	Updated prebinding guidelines.
2005-08-11	Explained how to set the preferred address for frameworks. Added information about what happens when prebinding fails.
2005-06-04	Updated prebinding information for OS X v10.4.
2005-04-29	Replaced Sampler examples with Shark examples. Document title changed. Old title was *Launch Time Performance*.
2004-05-27	Corrected prebinding address ranges.
2004-04-15	Updated address ranges pertaining to prebinding.
2003-07-25	Minor bug fixes for OS X v10.3.
2003-05-15	First revision of this programming topic. Some of the information appeared in the document *Inside OS X: Performance*.

Index

fs_usage 9, 14
sample 12
Shark 11

U

UpTime function 10